Program Coupling and Parallel Data Transfer Using PAWS

Sue Mniszewski, CCS-3

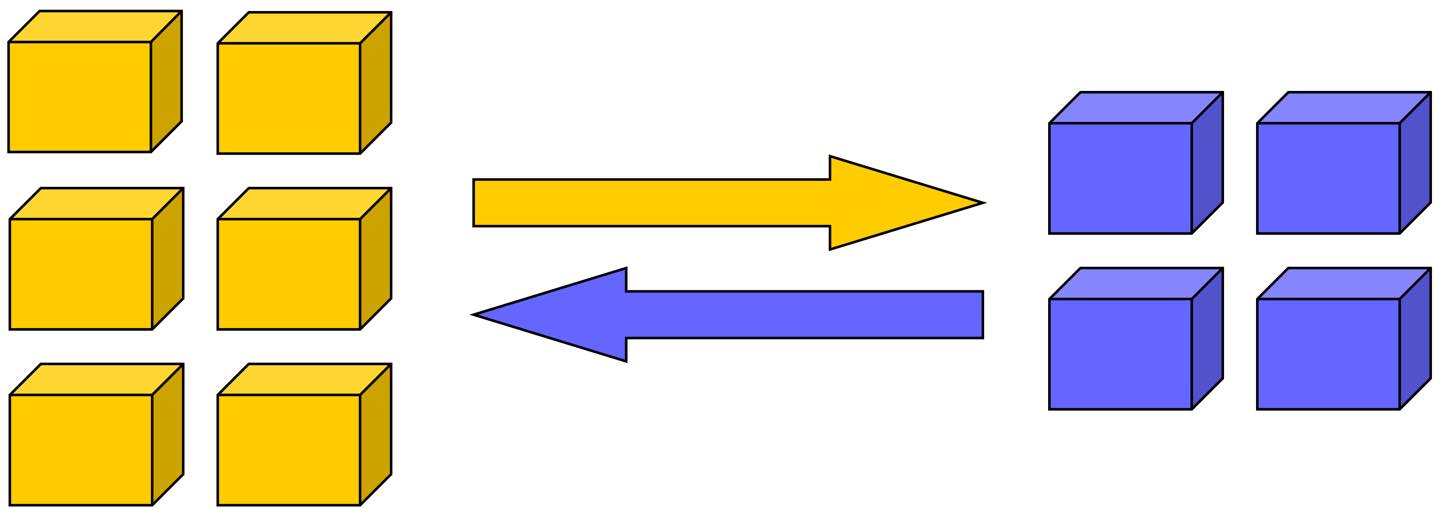
Pat Fasel, CCS-3

Craig Rasmussen, CCS-1

http://www.acl.lanl.gov/paws

Primary Goal of PAWS

- Provide the ability to share data between two separate, parallel applications, using the maximum bandwidth available



Challenges

- Coupling of parallel applications
- Avoid serialization
- Provide several synchronization strategies
- Connect parallel applications with unequal numbers of processes
- Be extensible to new, user-defined parallel distribution strategies

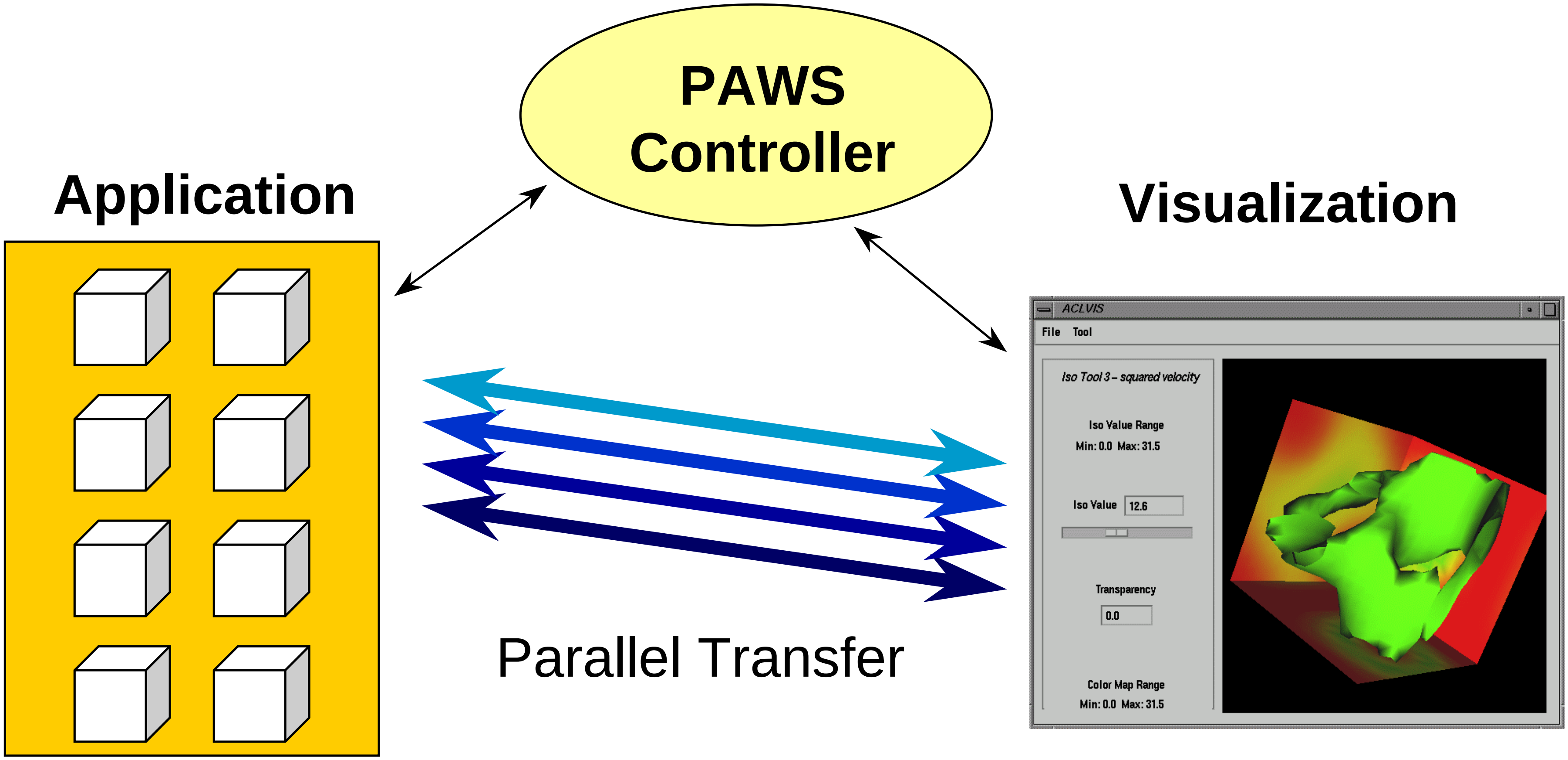
PAWS Design Goals

- Rewrite of PAWS 1
- Same general philosophy
- Using Cheetah/MPI run-time
- C++, C, and Fortran APIs
- Separation of layout and data (Ports & Views)
- Still single-threaded
- Channel abstraction
- Action caching (no barriers!)

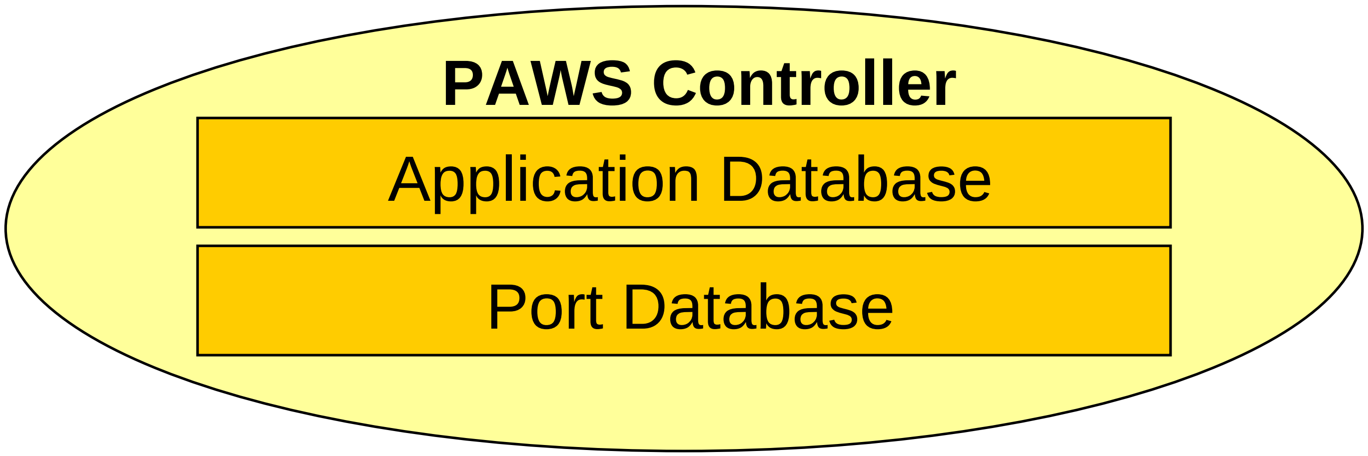
PAWS Port Concept

- Separates layout from data
- Has a shape, but no data type
- Ports are connected, not data
- Allows multiple datas to use a single port
- Scalar ports for int, float, double, string
- View ports for rectilinear parallel data

Sample PAWS Environment



PAWS Controller



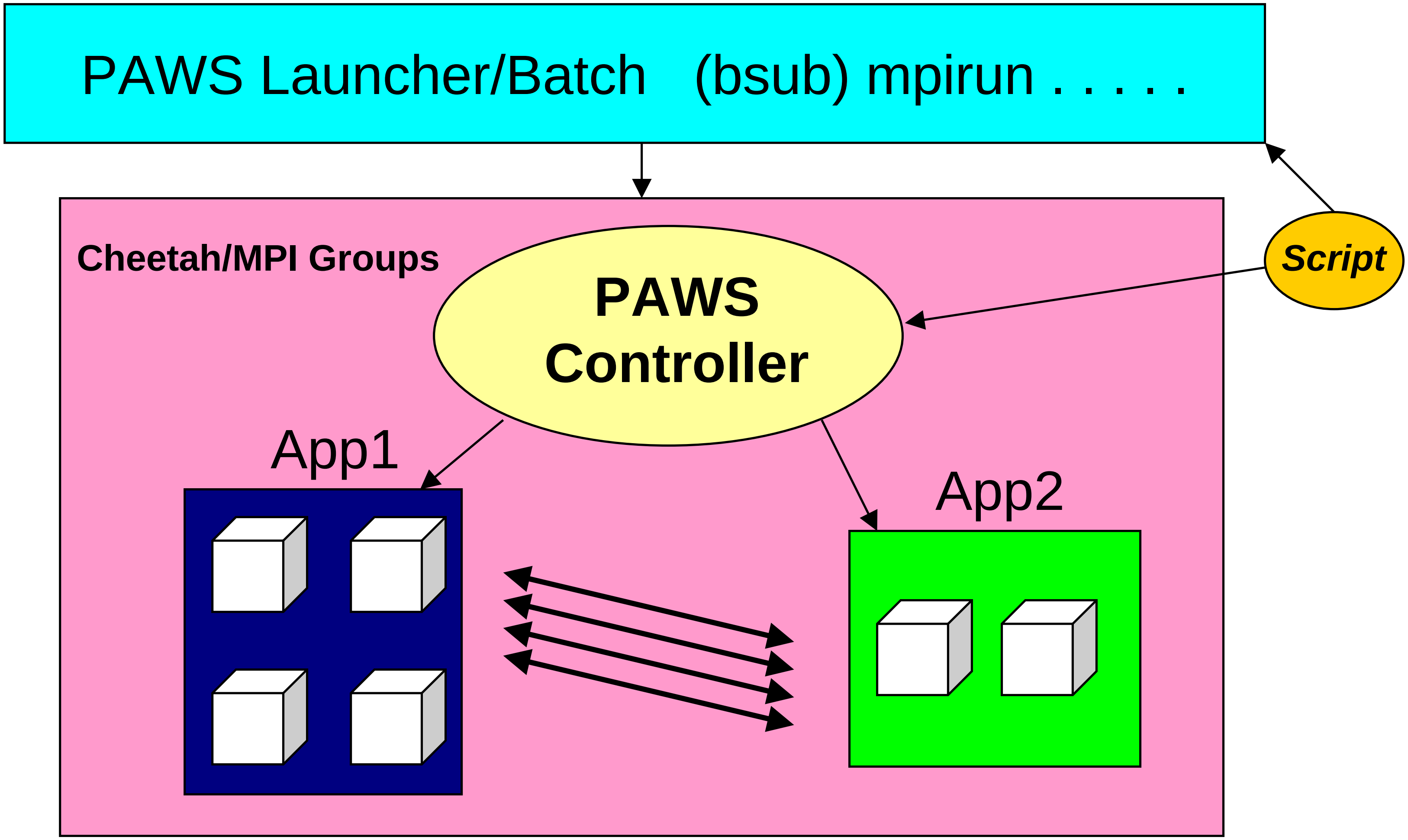
- Database with registered applications
- Database with registered ports
- Initialize connection between ports

PAWS Application

- Must be told where to find Controller
- Use PAWS API to:
  - Register app and ports with Controller
  - Query Controller databases
  - Make connections
  - Send/Receive data
  - Disconnect ports or app from Controller

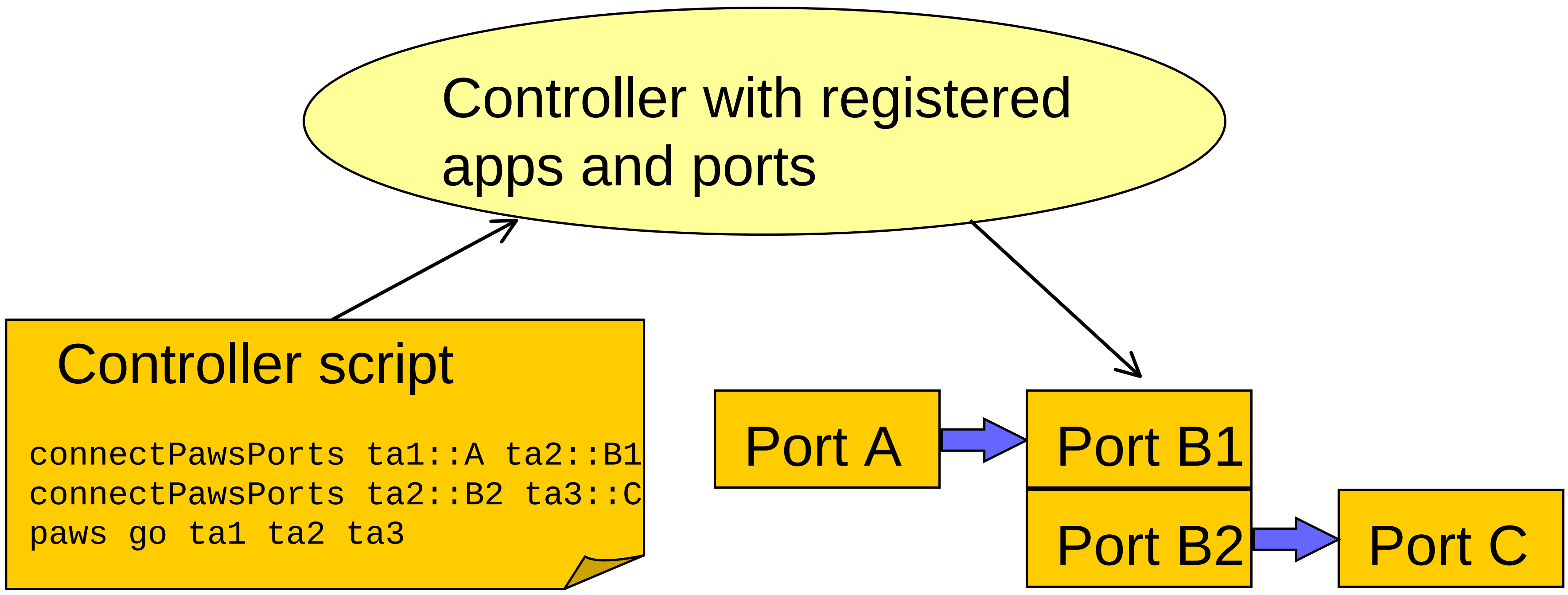
PARALLEL APPLICATION WORK SPACE

Running Applications



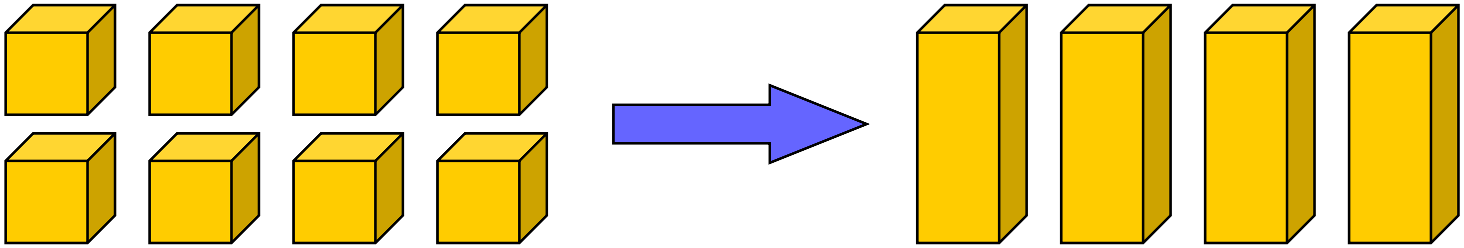
PAWS Networks

- PAWS Launcher/Controller accepts a script to start apps and connect ports



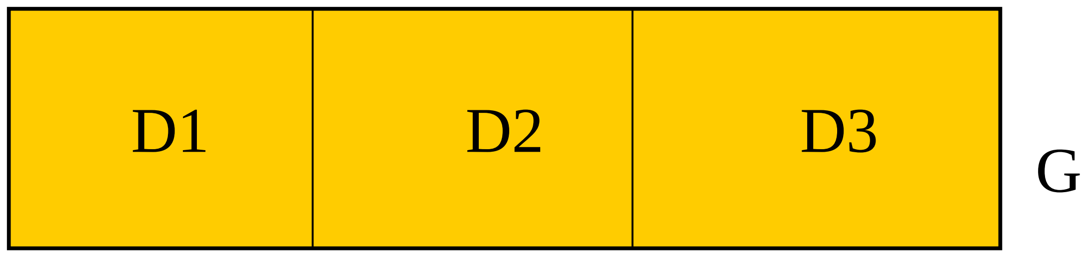
Parallel Redistribution

- Each application can partition data differently
  - Different numbers of processes
  - Different parallel layouts



PAWS Data Model for Rectilinear Data

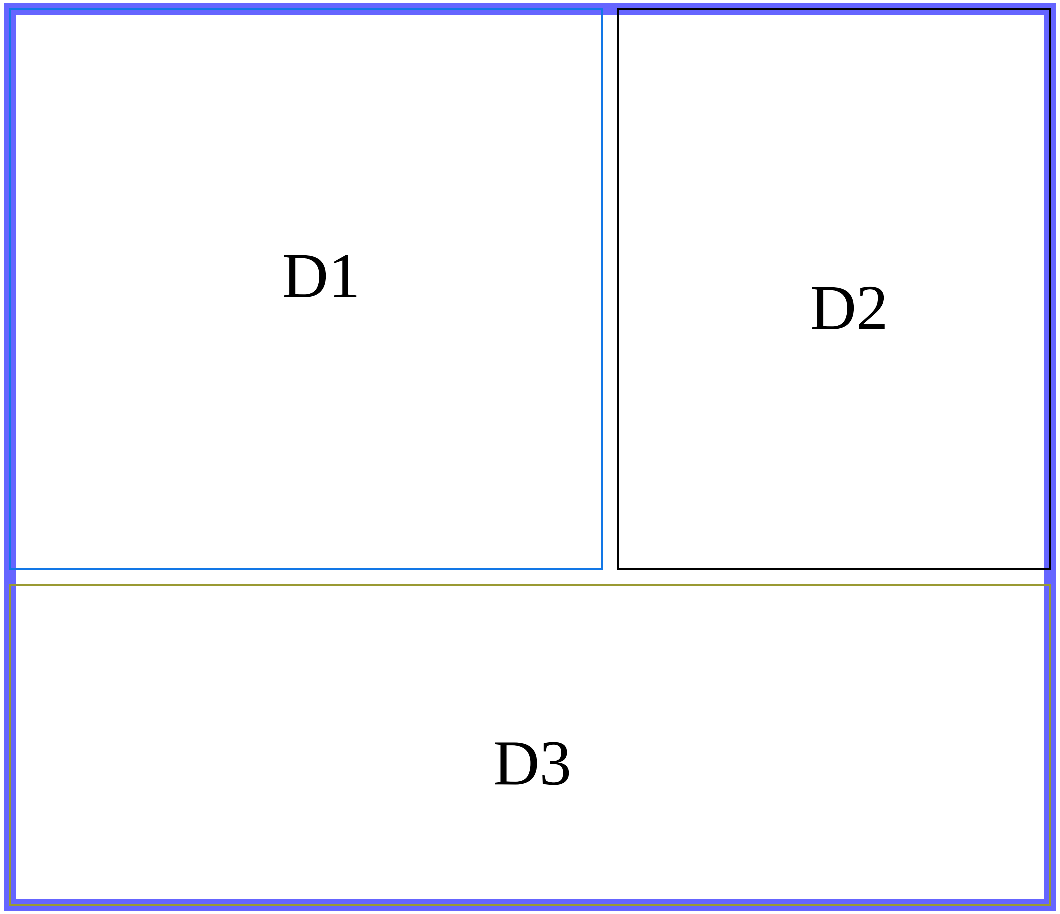
- Parallel data objects in PAWS have two parts:
  - A global domain G, and a list of subdomains { Di }



  - Allocated blocks of memory, and mappings (known as *views)* from memory to the { Di } subdomains
- G and { Di } are all that is needed to compute a *message schedule*
- Views and allocated memory locations can change

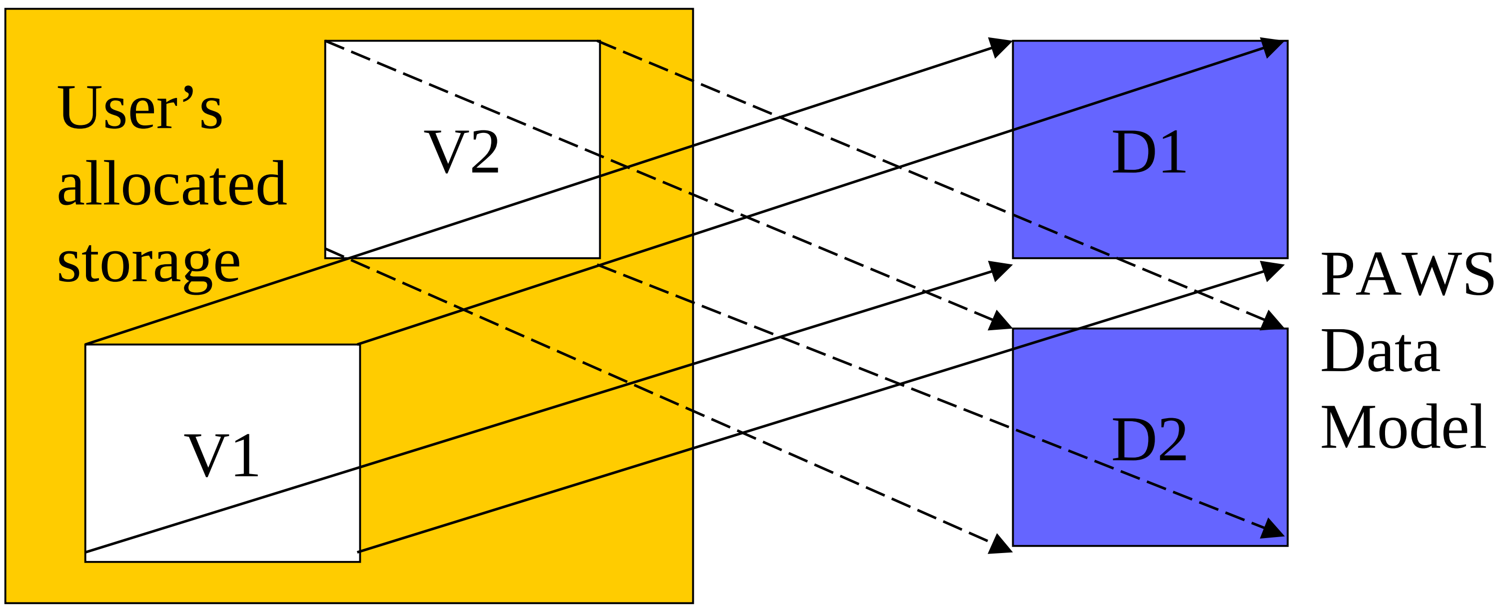
Example of G and { Di } in PAWS

G



PAWS Data Model: Views

- Views describe how data is mapped from user storage to the subdomain blocks:



PAWS Data Model: Types of Views

- Ways that we can specify the view domain Vi:
  - Strided “normal” D-dimensional domain, zero-based
    - 1D example: Vi = {0, Asize-1, 1} means all of Ai.
  - Indirection list: list of M indirection points, each a D-dimensional zero-based index into Ai domain. (future)
    - 1D example: Vi = { {3,4}, {8,1}, {4,4} } is a 3-element list.
  - Function view: generate the data from a user-provided function (or functor), instead of copying from memory. (future)
  - The user could implement his/her own type of view. This will be the user-extensibility method in PAWS.

PAWS Data: Row- or Column-major

- Organization of data in memory (row- or column-major) is a characteristic of the data storage, not the data model.
- Data model always stores sizes in same format. Dimension information is listed left-to-right, first dimension to last.
- Only use row- or column-major info to calculate final location in memory during transfers.
- So, R- or C-major is a characteristic of a *view*. Specify it when you specify a view.

Registering a Port

- Information specified when you <u>register</u> ports from the user's program:
  - Initial data model info G and { Di } (data model)
  - Input or Output
  - Port type, scalar or distributed

Connecting Ports

- What info does both sides of a connection share?
  - Data model info G and { Di } for both sides
  - Dynamic/static nature of layout on both sides
  - Send/Receive message schedules, if they can be computed

Synchronous Data Transfer

- Data is transferred when both sides indicate they are ready
  - One side does a send(), the other a receive()
- Before transfer, users must describe G, { Di }, and the assignment of each Di to the processors when registering a Port.
- The user provides the views, Vi, and pointers to the allocated blocks of memory, in the send() and receive() calls.

```
//        REGISTER WITH PAWS
   Paws::Application app(argc, argv);

//        CREATE PORT
   Paws::Representation rep(wholeDom, my
   Paws::Port A("A", rep, Paws::PAWS_OU
   Paws::Port I("I", Paws::PAWS_OUT, Pav
   app.addPort(&I); app.addPort(&A);

//        ALLOCATE DATA AND CREATE V
   int* data = new int[mySize];
   Paws::ViewArray<int> view(Paws::PAWS
   view.addViewBlock(data, myAllocDom, n

//        READY
   app.ready();

//        SEND DATA
   int iter = 100;
   I.send(&iter);
   for (i = 0; i < iter; i++) {
      A.send(view);
   }

//        CLEANUP
   app.finalize();
```

```
/*    REGISTER WITH PAWS */
 paws_initialize(argc, argv);
/*    CREATE PORTS */
   int rep = paws_representation(wholeDom, myDom, myRank);
   int B = paws_view_port("B", rep, PAWS_IN);
   int J = paws_scalar_port("J", PAWS_IN);
/*    ALLOCATE DATA AND CREATE VIEW  */
   int* data = (int*) malloc(mySize * sizeof(int));
   int view = paws_view(PAWS_INTEGER, PAWS_ROW);
   paws_add_view_block(view, data, myAllocDom, myViewDom);
/*    READY */
   paws_ready();
/*    RECEIVE DATA */
   paws_scalar_receive(J, iter, PAWS_INTEGER, 1);
   for (i = 0; i < iter; i++) {
      paws_view_receive(B, view);
   }
/*    CLEANUP */
paws_finalize();
```

Dynamically Resizing Data

- What if one side decided to resize or repartition?
  - The other side must find out about new size or distribution of the other side
  - A new message schedule must be calculated
  - Memory may have to be reallocated
- PAWS provides an interface that:
  - Lets the user find out the new global domain G and subdomains { Di } for the other side of the connection
  - Lets the user reallocate their data if it is necessary
  - Supports resize events from either side of the connection

PARALLEL APPLICATION WORK SPACE

Resize Code Example

Send

```
Application app(argc, argv);

//    CREATE EMPTY REP AND VIEW
Representation rep(myRank);
Port A("A", rep, PAWS_OUT, PAWS_DISTRI
ViewArray<int> view(PAWS_COLUMN);

app.ready();

//    RESIZE DATA ON EVERY SEND
for (i = 0; i < iter; i++) {
    int* data = new int[newMySize];
    rep.update(newWholeDom, newMyDom);
    view.update(data, newMyAllocDom, newMy
    A.resize(rep);
    A.send(view);
}
app.finalize();
```

Receive

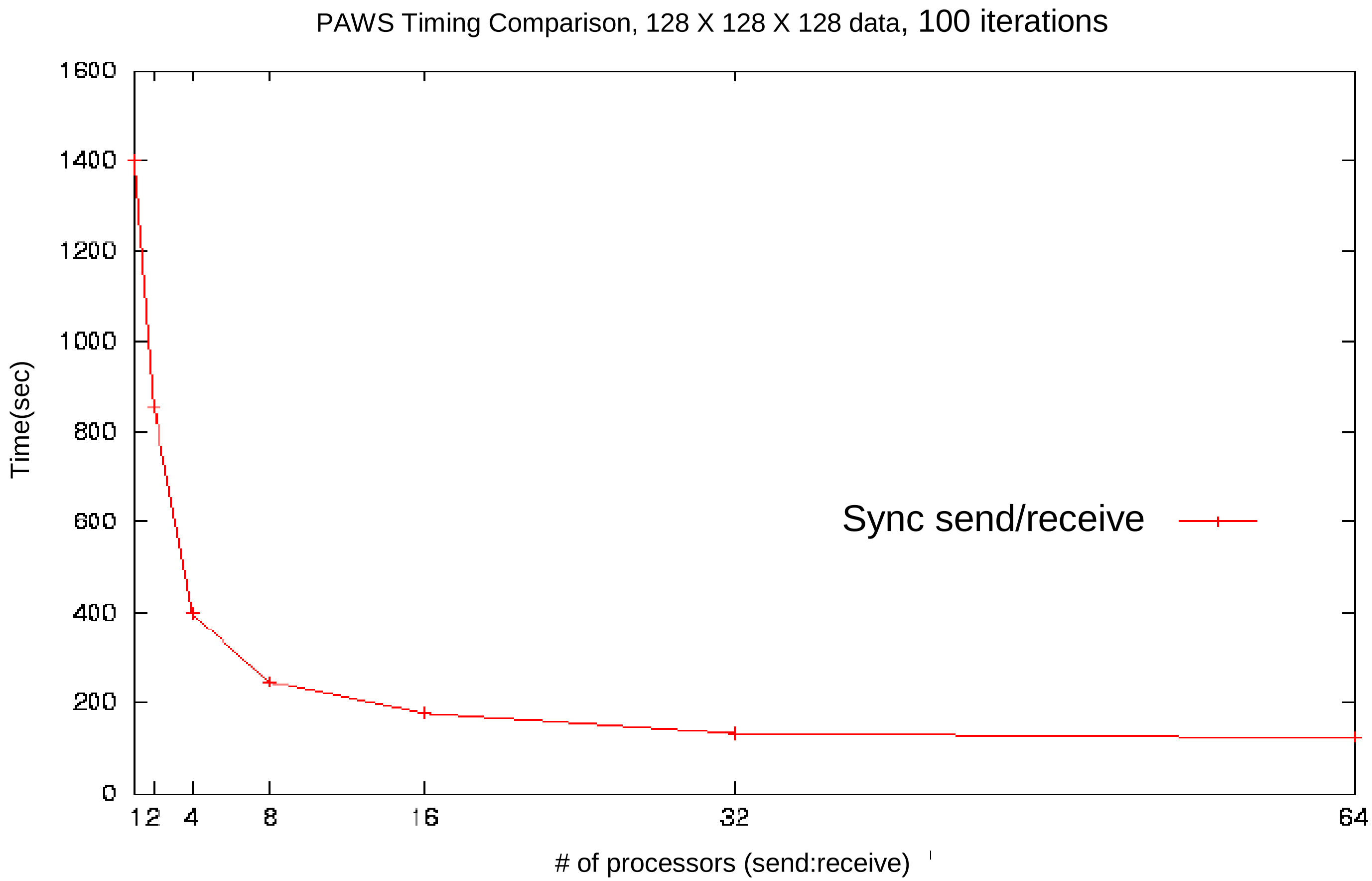
```
Application app(argc, argv);

//    CREATE EMPTY REP AND VIEW
Representation rep(myRank);
Port B("B", rep, PAWS_IN, PAWS_DISTRIBUTED);
ViewArray<int> view(PAWS_COLUMN);

app.ready();

//    WAIT FOR SIZE ON EVERY RECEIVE
for (i = 0; i < iter; i++)
    B.resizeWait();
    newWholeDom = B.domain();
    newMyDom = redistribute(newWholeDom, rank);
    int* data = new int[newMySize];
    rep.update(newWholeDom, newMyDom);
    view.update(data, newMyAllocDom, newMyViewDom);
    B.update(rep);
    B.receive(view);
}
app.finalize();
```

Preliminary Results



Future Directions

- Asynchronous data transfer
  - get(), unlock(), lock(), wait(), check()
- MPI 2 Communications
- More parallel data structures
  - Particles, unstructured, trees, etc.
- Better scripting interface
- Common Component Architecture (CCA) style framework